**City-Dataset:**[**https://docs.google.com/spreadsheets/d/1dk9kRwcMxj5USuJqxtITD05S-aOUD6fzNzV W41dcpgc/edit?usp=sharing**](https://docs.google.com/spreadsheets/d/1dk9kRwcMxj5USuJqxtITD05S-aOUD6fzNzVW41dcpgc/edit?usp=sharing)

**Q1**. Query all columns for all American cities in the CITY table with populations larger than 100000.

The CountryCode for America is USA.

The CITY table is described as follows:



**Query**: SELECT \* FROM CITY WHERE COUNTRYCODE = 'USA' AND POPULATION > 100000;

**Q2**. Query the NAME field for all American cities in the CITY table with populations larger than 120000. The CountryCode for America is USA.

The CITY table is described as follows:



**Query**: SELECT NAME FROM CITY WHERE COUNTRYCODE = 'USA' AND POPULATION > 120000;

**Q3.** Query all columns (attributes) for every row in the CITY table. The CITY table is described as follows:



**Query**: SELECT \*FROM CITY;

**Q4**. Query all columns for a city in CITY with the ID 1661. The CITY table is described as follows:



**Query**: SELECT \*FROM CITY WHERE ID = 1661;

**Q5**. Query all attributes of every Japanese city in the CITY table. The COUNTRYCODE for Japan is JPN.

The CITY table is described as follows:



**Q6.** Query the names of all the Japanese cities in the CITY table. The COUNTRYCODE for Japan is JPN.

The CITY table is described as follows:



**Query**: SELECT \*FROM CITY WHERE COUNTRYCODE = 'JPN';

**station-table:**[https://docs.google.com/spreadsheets/d/1sHPhE7walQD5mL7ppFNqybyoOJY3E51N0 cWYzhp2UH4/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1sHPhE7walQD5mL7ppFNqybyoOJY3E51N0cWYzhp2UH4/edit?usp=sharing)

**Q7.** Query a list of CITY and STATE from the STATION table. The STATION table is described as follows:



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Query**: SELECT CITY, STATE FROM STATION;

**Q8.** Query a list of CITY names from STATION for cities that have an even ID number. Print the results in any order, but exclude duplicates from the answer. The STATION table is described as follows:



where LAT\_N is the northern latitude and LONG\_W is the western longitude

**Query**: SELECT DISTINCT CITY FROM STATION WHERE MOD(ID, 2) = 0;

**Q9**. Find the difference between the total number of CITY entries in the table and the number of distinct CITY entries in the table.

The STATION table is described as follows:



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

For example, if there are three records in the table with CITY values 'New York', 'New York', 'Bengalaru', there are 2 different city names: 'New York' and 'Bengalaru'. The query returns , because total number of records - number of unique city names = 3-2 =1

**Query**: SELECT COUNT(\*) - COUNT(DISTINCT CITY) AS Difference FROM STATION;

**Q10.** Query the two cities in STATION with the shortest and longest CITY names, as well as their respective lengths (i.e.: number of characters in the name). If there is more than one smallest or largest city, choose the one that comes first when ordered alphabetically. The STATION table is described as follows:



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

Sample Input

For example, CITY has four entries: DEF, ABC, PQRS and WXY.

Sample Output

ABC 3

PQRS 4

**Hint -**

When ordered alphabetically, the CITY names are listed as ABC, DEF, PQRS, and WXY, with lengths and. The longest name is PQRS, but there are options for shortest named city. Choose ABC, because it comes first alphabetically.

Note

You can write two separate queries to get the desired output. It need not be a single query.

For the city with the shortest name:

SELECT CITY, LENGTH(CITY) AS NameLength

FROM STATION

ORDER BY LENGTH(CITY), CITY

LIMIT 1;

--This query selects the CITY column and calculates the length of the CITY name using the ‘LENGTH’ Function. It orders the results by the length of the CITY name is ascending order and then alphabetically, and limits the result to only the first row.

For the city with the longest name:

SELECT CITY, LENGTH(CITY) AS NameLength

FROM STATION

ORDER BY LENGTH(CITY) DESC, CITY

LIMIT 1;

**Q11**. Query the list of CITY names starting with vowels (i.e., a, e, i, o, or u) from STATION. Your result cannot contain duplicates.

Input Format

The STATION table is described as follows:



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Query**: SELECT DISTINCT CITY

FROM STATION

WHERE CITY REGEXP '^[aeiouAEIOU]';

Or

**Query2**: SELECT DISTINCT CITY

FROM STATION

WHERE CITY LIKE '[aeiouAEIOU]%';

**Q12.** Query the list of CITY names ending with vowels (a, e, i, o, u) from STATION. Your result cannot contain duplicates.

Input Format

The STATION table is described as follows:



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Query1:** SELECT DISTINCT CITY

FROM STATION

WHERE CITY REGEXP '[aeiouAEIOU]$';

Or

**Query2**: SELECT DISTINCT CITY

FROM STATION

WHERE CITY LIKE '%[aeiouAEIOU]';

**Q13.** Query the list of CITY names from STATION that do not start with vowels. Your result cannot contain duplicates.

Input Format

The STATION table is described as follows:



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Query 1:** SELECT DISTINCT CITY

FROM STATION

WHERE CITY NOT REGEXP '^[aeiouAEIOU]';

Or

**Query2**: SELECT DISTINCT CITY

FROM STATION

WHERE CITY NOT LIKE '[aeiouAEIOU]%';

**Q14.** Query the list of CITY names from STATION that do not end with vowels. Your result cannot contain duplicates.

Input Format

The STATION table is described as follows:



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Query 1**: SELECT DISTINCT CITY

FROM STATION

WHERE CITY NOT REGEXP '[aeiouAEIOU]$';

Or

**Query2**: SELECT DISTINCT CITY

FROM STATION

WHERE CITY NOT LIKE '%[aeiouAEIOU]';

**Q15.** Query the list of CITY names from STATION that either do not start with vowels or do not end with vowels. Your result cannot contain duplicates.

Input Format

The STATION table is described as follows:



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Query1**: SELECT DISTINCT CITY

FROM STATION

WHERE CITY NOT REGEXP '^[aeiouAEIOU]' OR CITY NOT REGEXP '[aeiouAEIOU]$';

or

**Query2**: SELECT DISTINCT CITY

FROM STATION

WHERE CITY NOT LIKE '[aeiouAEIOU]%' OR CITY NOT LIKE '%[aeiouAEIOU]';

**Q16.** Query the list of CITY names from STATION that do not start with vowels and do not end with vowels. Your result cannot contain duplicates.

Input Format

The STATION table is described as follows:



where LAT\_N is the northern latitude and LONG\_W is the western longitude.

**Query1:** SELECT DISTINCT CITY

FROM STATION

WHERE CITY NOT REGEXP '^[aeiouAEIOU].\*[aeiouAEIOU]$';

Or

**Query2**: SELECT DISTINCT CITY

FROM STATION

WHERE CITY NOT LIKE '[aeiouAEIOU]%[aeiouAEIOU]';

**Q17.**

Table: Product

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| product\_name | varchar |
| unit\_price | int |

product\_id is the primary key of this table.

Each row of this table indicates the name and the price of each product.

Table: Sales

|  |  |
| --- | --- |
| Column Name | Type |
| seller\_id | int |
| product\_id | int |
| buyer\_id | int |
| sale\_date | date |
| quantity | int |
| price | int |

This table has no primary key, it can have repeated rows.

product\_id is a foreign key to the Product table.

Each row of this table contains some information about one sale.

Write an SQL query that reports the products that were only sold in the first quarter of 2019. That is, between 2019-01-01 and 2019-03-31 inclusive.

Return the result table in any order.

The query result format is in the following example.

Input:

Product table:

|  |  |  |
| --- | --- | --- |
| product\_id | product\_name | unit\_price |
| 1 | S8 | 1000 |
| 2 | G4 | 800 |
| 3 | iPhone | 1400 |

Sales table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| seller\_id | product\_id | buyer\_id | sale\_date | quantity | price |
| 1 | 1 | 1 | 2019-01-21 | 2 | 2000 |
| 1 | 2 | 2 | 2019-02-17 | 1 | 800 |
| 2 | 2 | 3 | 2019-06-02 | 1 | 800 |
| 3 | 3 | 4 | 2019-05-13 | 2 | 2800 |

Output:

|  |  |
| --- | --- |
| product\_id | product\_name |
| 1 | S8 |

Explanation:

The product with id 1 was only sold in the spring of 2019.

The product with id 2 was sold in the spring of 2019 but was also sold after the spring of 2019.

The product with id 3 was sold after spring 2019.

We return only product 1 as it is the product that was only sold in the spring of 2019.

**Query**: SELECT p.product\_id, p.product\_name

FROM Product p

WHERE p.product\_id NOT IN (

SELECT s.product\_id

FROM Sales s

WHERE s.sale\_date >= '2019-01-01' AND s.sale\_date <= '2019-03-31'

)

**Q18.**

Table: Views

|  |  |
| --- | --- |
| Column Name | Type |
| article\_id | int |
| author\_id | int |
| viewer\_id | int |
| view\_date | date |

There is no primary key for this table, it may have duplicate rows.

Each row of this table indicates that some viewer viewed an article (written by some author) on some date.

Note that equal author\_id and viewer\_id indicate the same person.

Write an SQL query to find all the authors that viewed at least one of their own articles.

Return the result table sorted by id in ascending order. The query result format is in the following example.

Input:

Views table:

|  |  |  |  |
| --- | --- | --- | --- |
| article\_id | author\_id | viewer\_id | view\_date |
| 1 | 3 | 5 | 2019-08-01 |
| 1 | 3 | 6 | 2019-08-02 |
| 2 | 7 | 7 | 2019-08-01 |
| 2 | 7 | 6 | 2019-08-02 |
| 4 | 7 | 1 | 2019-07-22 |
| 3 | 4 | 4 | 2019-07-21 |
| 3 | 4 | 4 | 2019-07-21 |

Output:

|  |
| --- |
| id |
| 4 |
| 7 |

**Query:** SELECT DISTINCT author\_id AS id

FROM Views

WHERE author\_id = viewer\_id

ORDER BY id ASC;

**Q19.**

Table: Delivery

|  |  |
| --- | --- |
| Column Name | Type |
| delivery\_id | int |
| customer\_id | int |
| order\_date | date |
| customer\_pref\_delivery\_date | date |

delivery\_id is the primary key of this table.

The table holds information about food delivery to customers that make orders at some date and specify a preferred delivery date (on the same order date or after it).

If the customer's preferred delivery date is the same as the order date, then the order is called immediately; otherwise, it is called scheduled.

Write an SQL query to find the percentage of immediate orders in the table, rounded to 2 decimal places.

The query result format is in the following example.

Input:

Delivery table:

|  |  |  |  |
| --- | --- | --- | --- |
| delivery\_id | customer\_id | order\_date | customer\_pref\_ delivery\_date |
| 1 | 1 | 2019-08-01 | 2019-08-02 |
| 2 | 5 | 2019-08-02 | 2019-08-02 |
| 3 | 1 | 2019-08-11 | 2019-08-11 |
| 4 | 3 | 2019-08-24 | 2019-08-26 |
| 5 | 4 | 2019-08-21 | 2019-08-22 |
| 6 | 2 | 2019-08-11 | 2019-08-13 |

Output:

|  |
| --- |
| immediate\_percentage |
| 33.33 |

Explanation: The orders with delivery id 2 and 3 are immediate while the others are scheduled.

**Query**: SELECT ROUND((COUNT(CASE WHEN order\_date = customer\_pref\_delivery\_date THEN 1 END) / COUNT(\*) \* 100), 2) AS immediate\_percentage FROM Delivery;

**Q20.**

Table: Ads

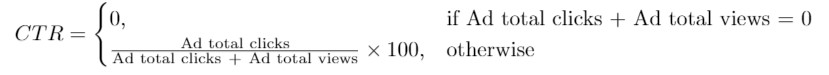
|  |  |
| --- | --- |
| Column Name | Type |
| ad\_id | int |
| user\_id | int |
| action | enum |

(ad\_id, user\_id) is the primary key for this table.

Each row of this table contains the ID of an Ad, the ID of a user, and the action taken by this user regarding this Ad.

The action column is an ENUM type of ('Clicked', 'Viewed', 'Ignored').

A company is running Ads and wants to calculate the performance of each Ad. Performance of the Ad is measured using Click-Through Rate (CTR) where:



Write an SQL query to find the ctr of each Ad. Round ctr to two decimal points.

Return the result table ordered by ctr in descending order and by ad\_id in ascending order in case of a tie.

The query result format is in the following example.

Input:

Ads table:

|  |  |  |
| --- | --- | --- |
| ad\_id | user\_id | action |
| 1 | 1 | Clicked |
| 2 | 2 | Clicked |
| 3 | 3 | Viewed |
| 5 | 5 | Ignored |
| 1 | 7 | Ignored |
| 2 | 7 | Viewed |
| 3 | 5 | Clicked |
| 1 | 4 | Viewed |
| 2 | 11 | Viewed |
| 1 | 2 | Clicked |

Output:

|  |  |
| --- | --- |
| ad\_id | ctr |
| 1 | 66.67 |
| 3 | 50 |
| 2 | 33.33 |
| 5 | 0 |

Explanation:

for ad\_id = 1, ctr = (2/(2+1)) \* 100 = 66.67 for ad\_id = 2, ctr = (1/(1+2)) \* 100 = 33.33 for ad\_id = 3, ctr = (1/(1+1)) \* 100 = 50.00 for ad\_id = 5, ctr = 0.00, Note that ad\_id = 5 has no clicks or views. Note that we do not care about Ignored Ads.

**Query**: SELECT ad\_id,

ROUND((IFNULL(SUM(CASE WHEN action = 'Clicked' THEN 1 ELSE 0 END), 0) /

(IFNULL(SUM(CASE WHEN action = 'Clicked' THEN 1 ELSE 0 END) +

SUM(CASE WHEN action = 'Viewed' THEN 1 ELSE 0 END), 0)) \* 100), 2) AS ctr

FROM Ads

GROUP BY ad\_id

ORDER BY ctr DESC, ad\_id ASC;

**Q21.**

Table: Employee

|  |  |
| --- | --- |
| Column Name | Type |
| employee\_id | int |
| team\_id | int |

employee\_id is the primary key for this table.

Each row of this table contains the ID of each employee and their respective team.

Write an SQL query to find the team size of each of the employees.

Return result table in any order.

The query result format is in the following example.

Input:

Employee Table:

|  |  |
| --- | --- |
| employee\_id | team\_id |
| 1 | 8 |
| 2 | 8 |
| 3 | 8 |
| 4 | 7 |
| 5 | 9 |
| 6 | 9 |

Output:

|  |  |
| --- | --- |
| employee\_id | team\_size |
| 1 | 3 |
| 2 | 3 |
| 3 | 3 |
| 4 | 1 |
| 5 | 2 |
| 6 | 2 |

Explanation:

Employees with Id 1,2,3 are part of a team with team\_id = 8.

An employee with Id 4 is part of a team with team\_id = 7.

Employees with Id 5,6 are part of a team with team\_id = 9.

**Query**: SELECT employee\_id, COUNT(\*) AS team\_size

FROM Employee

GROUP BY employee\_id;

**Q22.**

Table: Countries

|  |  |
| --- | --- |
| Column Name | Type |
| country\_id | int |
| country\_name | varchar |

country\_id is the primary key for this table.

Each row of this table contains the ID and the name of one country.

Table: Weather

|  |  |
| --- | --- |
| Column Name | Type |
| country\_id | int |
| weather\_state | int |
| day | date |

(country\_id, day) is the primary key for this table.

Each row of this table indicates the weather state in a country for one day.

Write an SQL query to find the type of weather in each country for November 2019.

The type of weather is:

● Cold if the average weather\_state is less than or equal 15, ● Hot if the average weather\_state is greater than or equal to 25, and ● Warm otherwise.

Return result table in any order.

The query result format is in the following example.

Input:

Countries table:

|  |  |
| --- | --- |
| country\_id | country\_name |
| 2 | USA |
| 3 | Australia |
| 7 | Peru |
| 5 | China |
| 8 | Morocco |
| 9 | Spain |

Weather table:

|  |  |  |
| --- | --- | --- |
| country\_id | weather\_state | day |
| 2 | 15 | 2019-11-01 |
| 2 | 12 | 2019-10-28 |
| 2 | 12 | 2019-10-27 |
| 3 | -2 | 2019-11-10 |
| 3 | 0 | 2019-11-11 |
| 3 | 3 | 2019-11-12 |
| 5 | 16 | 2019-11-07 |
| 5 | 18 | 2019-11-09 |
| 5 | 21 | 2019-11-23 |
| 7 | 25 | 2019-11-28 |
| 7 | 22 | 2019-12-01 |
| 7 | 20 | 2019-12-02 |
| 8 | 25 | 2019-11-05 |
| 8 | 27 | 2019-11-15 |
| 8 | 31 | 2019-11-25 |
| 9 | 7 | 2019-10-23 |
| 9 | 3 | 2019-12-23 |

Output:

|  |  |
| --- | --- |
| country\_name | weather\_type |
| USA | Cold |
| Australia | Cold |
| Peru | Hot |
| Morocco | Hot |
| China | Warm |

Explanation:

Average weather\_state in the USA in November is (15) / 1 = 15 so the weather type is Cold.

Average weather\_state in Australia in November is (-2 + 0 + 3) / 3 = 0.333 so the weather type is Cold.

Average weather\_state in Peru in November is (25) / 1 = 25 so the weather type is Hot.

The average weather\_state in China in November is (16 + 18 + 21) / 3 = 18.333 so the weather type is warm.

Average weather\_state in Morocco in November is (25 + 27 + 31) / 3 = 27.667 so the weather type is Hot.

We know nothing about the average weather\_state in Spain in November so we do not include it in the result table.

**Query**: SELECT c.country\_name,

CASE

WHEN AVG(w.weather\_state) <= 15 THEN 'Cold'

WHEN AVG(w.weather\_state) >= 25 THEN 'Hot'

ELSE 'Warm'

END AS weather\_type

FROM Countries c

INNER JOIN Weather w ON c.country\_id = w.country\_id

WHERE w.day >= '2019-11-01' AND w.day <= '2019-11-30'

GROUP BY c.country\_name;

**Q23.**

Table: Prices

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| start\_date | date |
| end\_date | date |
| price | int |

(product\_id, start\_date, end\_date) is the primary key for this table.

Each row of this table indicates the price of the product\_id in the period from start\_date to end\_date. For each product\_id there will be no two overlapping periods. That means there will be no two intersecting periods for the same product\_id.

Table: UnitsSold

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| purchase\_date | date |
| units | int |

There is no primary key for this table, it may contain duplicates.

Each row of this table indicates the date, units, and product\_id of each product sold.

Write an SQL query to find the average selling price for each product. average\_price should be rounded to 2 decimal places.

Return the result table in any order.

The query result format is in the following example.

Input:

Prices table:

|  |  |  |  |
| --- | --- | --- | --- |
| product\_id | start\_date | end\_date | price |
| 1 | 2019-02-17 | 2019-02-28 | 5 |
| 1 | 2019-03-01 | 2019-03-22 | 20 |
| 2 | 2019-02-01 | 2019-02-20 | 15 |
| 2 | 2019-02-21 | 2019-03-31 | 30 |

UnitsSold table:

|  |  |  |
| --- | --- | --- |
| product\_id | purchase\_date | units |
| 1 | 2019-02-25 | 100 |
| 1 | 2019-03-01 | 15 |
| 2 | 2019-02-10 | 200 |
| 2 | 2019-03-22 | 30 |

Output:

|  |  |
| --- | --- |
| product\_id | average\_price |
| 1 | 6.96 |
| 2 | 16.96 |

Explanation:

Average selling price = Total Price of Product / Number of products sold.

Average selling price for product 1 = ((100 \* 5) + (15 \* 20)) / 115 = 6.96 Average selling price for product 2 = ((200 \* 15) + (30 \* 30)) / 230 = 16.96 **Q24.**

Table: Activity

|  |  |
| --- | --- |
| Column Name | Type |
| player\_id | int |
| device\_id | int |
| event\_date | date |
| games\_played | int |

(player\_id, event\_date) is the primary key of this table.

This table shows the activity of players of some games.

Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on someday using some device.

Write an SQL query to report the first login date for each player.

Return the result table in any order.

The query result format is in the following example.

Input:

Activity table:

|  |  |  |  |
| --- | --- | --- | --- |
| player\_id | device\_id | event\_date | games\_played |
| 1 | 2 | 2016-03-01 | 5 |
| 1 | 2 | 2016-05-02 | 6 |
| 2 | 3 | 2017-06-25 | 1 |
| 3 | 1 | 2016-03-02 | 0 |
| 3 | 4 | 2018-07-03 | 5 |

Output:

|  |  |
| --- | --- |
| player\_id | first\_login |
| 1 | 2016-03-01 |
| 2 | 2017-06-25 |
| 3 | 2016-03-02 |

**Query:** SELECT us.product\_id,

ROUND(SUM(us.units \* p.price) / SUM(us.units), 2) AS average\_price

FROM UnitsSold us

JOIN Prices p ON us.product\_id = p.product\_id

AND us.purchase\_date >= p.start\_date

AND us.purchase\_date <= p.end\_date

GROUP BY us.product\_id;

**Q25.**

Table: Activity

|  |  |
| --- | --- |
| Column Name | Type |
| player\_id | int |
| device\_id | int |
| event\_date | date |
| games\_played | int |

(player\_id, event\_date) is the primary key of this table.

This table shows the activity of players of some games.

Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on someday using some device.

Write an SQL query to report the device that is first logged in for each player.

Return the result table in any order.

The query result format is in the following example.

Input:

Activity table:

|  |  |  |  |
| --- | --- | --- | --- |
| player\_id | device\_id | event\_date | games\_played |
| 1 | 2 | 2016-03-01 | 5 |
| 1 | 2 | 2016-05-02 | 6 |
| 2 | 3 | 2017-06-25 | 1 |
| 3 | 1 | 2016-03-02 | 0 |
| 3 | 4 | 2018-07-03 | 5 |

Output:

|  |  |
| --- | --- |
| player\_id | device\_id |
| 1 | 2 |
| 2 | 3 |
| 3 | 1 |

**Query:** SELECT player\_id, MIN(event\_date) AS first\_login, device\_id

FROM Activity

GROUP BY player\_id, device\_id;

**Q26.**

Table: Products

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| product\_name | varchar |
| product\_category | varchar |

product\_id is the primary key for this table.

This table contains data about the company's products.

Table: Orders

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| order\_date | date |
| unit | int |

There is no primary key for this table. It may have duplicate rows.

product\_id is a foreign key to the Products table. unit is the number of products ordered in order\_date.

Write an SQL query to get the names of products that have at least 100 units ordered in February 2020 and their amount.

Return result table in any order.

The query result format is in the following example.

Input:

Products table:

|  |  |  |
| --- | --- | --- |
| product\_id | product\_name | product\_catego ry |
| 1 | Leetcode Solutions | Book |
| 2 | Jewels of Stringology | Book |
| 3 | HP | Laptop |
| 4 | Lenovo | Laptop |
| 5 | Leetcode Kit | T-shirt |

Orders table:

|  |  |  |
| --- | --- | --- |
| product\_id | order\_date | unit |
| 1 | 2020-02-05 | 60 |
| 1 | 2020-02-10 | 70 |
| 2 | 2020-01-18 | 30 |
| 2 | 2020-02-11 | 80 |
| 3 | 2020-02-17 | 2 |
| 3 | 2020-02-24 | 3 |
| 4 | 2020-03-01 | 20 |
| 4 | 2020-03-04 | 30 |
| 4 | 2020-03-04 | 60 |
| 5 | 2020-02-25 | 50 |
| 5 | 2020-02-27 | 50 |
| 5 | 2020-03-01 | 50 |

Output:

|  |  |
| --- | --- |
| product\_name | unit |
| Leetcode Solutions | 130 |
| Leetcode Kit | 100 |

Explanation:

Products with product\_id = 1 is ordered in February a total of (60 + 70) = 130.

Products with product\_id = 2 is ordered in February a total of 80.

Products with product\_id = 3 is ordered in February a total of (2 + 3) = 5.

Products with product\_id = 4 was not ordered in February 2020.

Products with product\_id = 5 is ordered in February a total of (50 + 50) = 100.

**Query:** SELECT p.product\_name, SUM(o.unit) AS total\_units

FROM Products p

JOIN Orders o ON p.product\_id = o.product\_id

WHERE o.order\_date >= '2020-02-01' AND o.order\_date <= '2020-02-29'

GROUP BY p.product\_id, p.product\_name

HAVING SUM(o.unit) >= 100;

**Q27.**

Table: Users

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id | int |
| name | varchar |
| mail | varchar |

user\_id is the primary key for this table.

This table contains information of the users signed up in a website. Some emails are invalid.

Write an SQL query to find the users who have valid emails.

A valid e-mail has a prefix name and a domain where:

* The prefix name is a string that may contain letters (upper or lower case), digits, underscore '\_', period '.', and/or dash '-'. The prefix name must start with a letter.
* The domain is '@leetcode.com'.

Return the result table in any order.

The query result format is in the following example.

Input:

Users table:

|  |  |  |
| --- | --- | --- |
| user\_id | name | mail |
| 1 | Winston | winston@leetc ode.com |
| 2 | Jonathan | jonathanisgreat |
| 3 | Annabelle | bella-@leetcod e.com |
| 4 | Sally | sally.come@lee tcode.com |
| 5 | Marwan | quarz#2020@le etcode.com |
| 6 | David | david69@gmail .com |
| 7 | Shapiro | .shapo@leetco de.com |

Output:

|  |  |  |
| --- | --- | --- |
| user\_id | name | mail |
| 1 | Winston | winston@leetc ode.com |
| 3 | Annabelle | bella-@leetcod e.com |
| 4 | Sally | sally.come@lee tcode.com |

Explanation:

The mail of user 2 does not have a domain.

The mail of user 5 has the # sign which is not allowed. The mail of user 6 does not have the leetcode domain. The mail of user 7 starts with a period.

**Query1:** SELECT user\_id, name, mail

FROM Users

WHERE mail REGEXP '^[a-zA-Z][a-zA-Z0-9\_.-][\*@leetcode\\.com$](mailto:*@leetcode\\.com$)';

Or

**Query2:** SELECT user\_id, name, mail

FROM Users

WHERE mail LIKE '[a-zA-Z]%@leetcode.com'

AND mail NOT LIKE '%#%'

AND mail NOT LIKE '%.@leetcode.com';

**Q28**.

Table: Customers

|  |  |
| --- | --- |
| Column Name | Type |
| customer\_id | int |
| name | varchar |
| country | varchar |

customer\_id is the primary key for this table.

This table contains information about the customers in the company.

Table: Product

|  |  |
| --- | --- |
| Column Name | Type |
| customer\_id | int |
| name | varchar |
| country | varchar |

product\_id is the primary key for this table.

This table contains information on the products in the company.

price is the product cost. Table: Orders

|  |  |
| --- | --- |
| Column Name | Type |
| order\_id | int |
| customer\_id | int |
| product\_id | int |
| order\_date | date |
| quantity | int |

order\_id is the primary key for this table.

This table contains information on customer orders.

customer\_id is the id of the customer who bought "quantity" products with id "product\_id". Order\_date is the date in format ('YYYY-MM-DD') when the order was shipped.

Write an SQL query to report the customer\_id and customer\_name of customers who have spent at least $100 in each month of June and July 2020.

Return the result table in any order.

The query result format is in the following example.

Input:

Customers table:

|  |  |  |
| --- | --- | --- |
| customer\_id | name | country |
| 1 | Winston | USA |
| 2 | Jonathan | Peru |
| 3 | Moustafa | Egypt |

Product table:

|  |  |  |
| --- | --- | --- |
| product\_id | description | price |
| 10 | LC Phone | 300 |
| 20 | LC T-Shirt | 10 |
| 30 | LC Book | 45 |
| 40 | LC Keychain | 2 |

Orders table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| order\_id | customer\_id | product\_id | order\_date | quantity |
| 1 | 1 | 10 | 2020-06-10 | 1 |
| 2 | 1 | 20 | 2020-07-01 | 1 |
| 3 | 1 | 30 | 2020-07-08 | 2 |
| 4 | 2 | 10 | 2020-06-15 | 2 |
| 5 | 2 | 40 | 2020-07-01 | 10 |
| 6 | 3 | 20 | 2020-06-24 | 2 |
| 7 | 3 | 30 | 2020-06-25 | 2 |
| 9 | 3 | 30 | 2020-05-08 | 3 |

Output:

|  |  |
| --- | --- |
| customer\_id | name |
| 1 | Winston |

Explanation:

Winston spent $300 (300 \* 1) in June and $100 ( 10 \* 1 + 45 \* 2) in July 2020.

Jonathan spent $600 (300 \* 2) in June and $20 ( 2 \* 10) in July 2020. Moustafa spent $110 (10 \* 2 + 45 \* 2) in June and $0 in July 2020.

**Query:** SELECT c.customer\_id, c.name

FROM Customers c

WHERE c.customer\_id IN (

SELECT o.customer\_id

FROM Orders o

JOIN Product p ON o.product\_id = p.product\_id

WHERE YEAR(o.order\_date) = 2020 AND MONTH(o.order\_date) IN (6, 7)

GROUP BY o.customer\_id, YEAR(o.order\_date), MONTH(o.order\_date)

HAVING SUM(p.price \* o.quantity) >= 100

AND COUNT(DISTINCT YEAR(o.order\_date), MONTH(o.order\_date)) = 2

);

**Q29**.

Table: TVProgram

|  |  |
| --- | --- |
| Column Name | Type |
| program\_date | date |
| content\_id | int |
| channel | varchar |

(program\_date, content\_id) is the primary key for this table. This table contains information about the programs on the TV. content\_id is the id of the program in some channel on the TV. Table: Content

|  |  |
| --- | --- |
| Column Name | Type |
| content\_id | varchar |
| title | varchar |
| Kids\_content | enum |
| content\_type | varchar |

content\_id is the primary key for this table.

Kids\_content is an enum that takes one of the values ('Y', 'N') where:

'Y' means content for kids, otherwise 'N' is not content for kids. content\_type is the category of the content as movies, series, etc.

Write an SQL query to report the distinct titles of the kid-friendly movies streamed in June 2020.

Return the result table in any order.

The query result format is in the following example.

Input:

TVProgram table:

|  |  |  |
| --- | --- | --- |
| program\_date | content\_id | channel |
| 2020-06-10 08:00 | 1 | LC-Channel |
| 2020-05-11 12:00 | 2 | LC-Channel |
| 2020-05-12 12:00 | 3 | LC-Channel |
| 2020-05-13 14:00 | 4 | Disney Ch |
| 2020-06-18 14:00 | 4 | Disney Ch |
| 2020-07-15 16:00 | 5 | Disney Ch |

Content table:

|  |  |  |  |
| --- | --- | --- | --- |
| content\_id | title | Kids\_content | content\_type |
| 1 | Leetcode Movie | N | Movies |
| 2 | Alg. for Kids | Y | Series |
| 3 | Database Sols | N | Series |
| 4 | Aladdin | Y | Movies |
| 5 | Cinderella | Y | Movies |

Output:

|  |
| --- |
| title |
| Aladdin |

Explanation:

"Leetcode Movie" is not a content for kids. "Alg. for Kids" is not a movie.

"Database Sols" is not a movie

"Alladin" is a movie, content for kids and was streamed in June 2020. "Cinderella" was not streamed in June 2020.

**Query:** SELECT DISTINCT c.title

FROM TVProgram t

JOIN Content c ON t.content\_id = c.content\_id

WHERE c.Kids\_content = 'Y' AND c.content\_type = 'Movies'

AND YEAR(t.program\_date) = 2020 AND MONTH(t.program\_date) = 6;

**Q30.**

Table: NPV

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| year | int |
| npv | int |

(id, year) is the primary key of this table.

The table has information about the id and the year of each inventory and the corresponding net present value.

Table: Queries

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| year | int |

(id, year) is the primary key of this table.

The table has information about the id and the year of each inventory query.

Write an SQL query to find the npv of each query of the Queries table.

Return the result table in any order.

The query result format is in the following example.

Input:

NPV table:

|  |  |  |
| --- | --- | --- |
| id | year | npv |
| 1 | 2018 | 100 |
| 7 | 2020 | 30 |
| 13 | 2019 | 40 |
| 1 | 2019 | 113 |
| 2 | 2008 | 121 |
| 3 | 2009 | 12 |
| 11 | 2020 | 99 |
| 7 | 2019 | 0 |

Queries table:

|  |  |
| --- | --- |
| id | year |
| 1 | 2019 |
| 2 | 2008 |
| 3 | 2009 |
| 7 | 2018 |
| 7 | 2019 |
| 7 | 2020 |
| 13 | 2019 |

Output:

|  |  |  |
| --- | --- | --- |
| id | year | npv |
| 1 | 2019 | 113 |
| 2 | 2008 | 121 |
| 3 | 2009 | 12 |
| 7 | 2018 | 0 |
| 7 | 2019 | 0 |
| 7 | 2020 | 30 |
| 13 | 2019 | 40 |

Explanation:

The npv value of (7, 2018) is not present in the NPV table, we consider it 0. The npv values of all other queries can be found in the NPV table.

**Query**: SELECT q.id, q.year, COALESCE(n.npv, 0) AS npv

FROM Queries q

LEFT JOIN NPV n ON q.id = n.id AND q.year = n.year;

**Q31.**

Table: NPV

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| year | int |
| npv | int |

(id, year) is the primary key of this table.

The table has information about the id and the year of each inventory and the corresponding net present value.

Table: Queries

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| year | int |

(id, year) is the primary key of this table.

The table has information about the id and the year of each inventory query.

Write an SQL query to find the npv of each query of the Queries table.

Return the result table in any order.

The query result format is in the following example.

Input:

NPV table:

|  |  |  |
| --- | --- | --- |
| id | year | npv |
| 1 | 2018 | 100 |
| 7 | 2020 | 30 |
| 13 | 2019 | 40 |
| 1 | 2019 | 113 |
| 2 | 2008 | 121 |
| 3 | 2009 | 12 |
| 11 | 2020 | 99 |
| 7 | 2019 | 0 |

Queries table:

|  |  |
| --- | --- |
| id | year |
| 1 | 2019 |
| 2 | 2008 |
| 3 | 2009 |
| 7 | 2018 |
| 7 | 2019 |
| 7 | 2020 |
| 13 | 2019 |

Output:

|  |  |  |
| --- | --- | --- |
| id | year | npv |
| 1 | 2019 | 113 |
| 2 | 2008 | 121 |
| 3 | 2009 | 12 |
| 7 | 2018 | 0 |
| 7 | 2019 | 0 |
| 7 | 2020 | 30 |
| 13 | 2019 | 40 |

Explanation:

The npv value of (7, 2018) is not present in the NPV table, we consider it 0. The npv values of all other queries can be found in the NPV table.

**Query**: SELECT q.id, q.year, COALESCE(n.npv, 0) AS npv

FROM Queries q

LEFT JOIN NPV n ON q.id = n.id AND q.year = n.year;

**Q32.**

Table: Employees

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |

id is the primary key for this table.

Each row of this table contains the id and the name of an employee in a company.

Table: EmployeeUNI

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| unique\_id | int |

(id, unique\_id) is the primary key for this table.

Each row of this table contains the id and the corresponding unique id of an employee in the company.

Write an SQL query to show the unique ID of each user, If a user does not have a unique ID replace just show null.

Return the result table in any order.

The query result format is in the following example.

Input:

Employees table:

|  |  |
| --- | --- |
| id | name |
| 1 | Alice |
| 7 | Bob |
| 11 | Meir |
| 90 | Winston |
| 3 | Jonathan |

EmployeeUNI table:

|  |  |
| --- | --- |
| id | unique\_id |
| 3 | 1 |
| 11 | 2 |
| 90 | 3 |

Output:

|  |  |
| --- | --- |
| unique\_id | name |
| null | Alice |
| null | Bob |
| 2 | Meir |
| 3 | Winston |
| 1 | Jonathan |

Explanation:

Alice and Bob do not have a unique ID, We will show null instead.

The unique ID of Meir is 2.

The unique ID of Winston is 3.

The unique ID of Jonathan is 1.

**Query**: SELECT e.id, e.name, eu.unique\_id

FROM Employees e

LEFT JOIN EmployeeUNI eu ON e.id = eu.id;

**Q33.**

Table: Users

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |

id is the primary key for this table. name is the name of the user. Table: Rides

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| user\_id | int |
| distance | int |

id is the primary key for this table. user\_id is the id of the user who travelled the distance "distance".

Write an SQL query to report the distance travelled by each user.

Return the result table ordered by travelled\_distance in descending order, if two or more users travelled the same distance, order them by their name in ascending order.

The query result format is in the following example.

Input:

Users table:

|  |  |
| --- | --- |
| id | name |
| 1 | Alice |
| 2 | Bob |
| 3 | Alex |
| 4 | Donald |
| 7 | Lee |
| 13 | Jonathan |
| 19 | Elvis |

Rides table:

|  |  |  |
| --- | --- | --- |
| id | user\_id | distance |
| 1 | 1 | 120 |
| 2 | 2 | 317 |
| 3 | 3 | 222 |
| 4 | 7 | 100 |
| 5 | 13 | 312 |
| 6 | 19 | 50 |
| 7 | 7 | 120 |
| 8 | 19 | 400 |
| 9 | 7 | 230 |

Output:

|  |  |
| --- | --- |
| name | travelled\_distan ce |
| Elvis | 450 |
| Lee | 450 |
| Bob | 317 |
| Jonathan | 312 |
| Alex | 222 |
| Alice | 120 |
| Donald | 0 |

Explanation:

Elvis and Lee travelled 450 miles, Elvis is the top traveller as his name is alphabetically smaller than Lee.

Bob, Jonathan, Alex, and Alice have only one ride and we just order them by the total distances of the ride.

Donald did not have any rides, the distance travelled by him is 0.

**Query**: SELECT u.name, COALESCE(SUM(r.distance), 0) AS travelled\_distance

FROM Users u

LEFT JOIN Rides r ON u.id = r.user\_id

GROUP BY u.name

ORDER BY travelled\_distance DESC, u.name ASC;

**Q34.**

Table: Products

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| product\_name | varchar |
| product\_category | varchar |

product\_id is the primary key for this table.

This table contains data about the company's products.

Table: Orders

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| order\_date | date |
| unit | int |

There is no primary key for this table. It may have duplicate rows.

product\_id is a foreign key to the Products table. unit is the number of products ordered in order\_date.

Write an SQL query to get the names of products that have at least 100 units ordered in February 2020 and their amount.

Return result table in any order.

The query result format is in the following example.

Input:

Products table:

|  |  |  |
| --- | --- | --- |
| product\_id | product\_name | product\_catego ry |
| 1 | Leetcode Solutions | Book |
| 2 | Jewels of Stringology | Book |
| 3 | HP | Laptop |
| 4 | Lenovo | Laptop |
| 5 | Leetcode Kit | T-shirt |

**Q35.**

Table: Movies

|  |  |
| --- | --- |
| Column Name | Type |
| movie\_id | int |
| title | varchar |

movie\_id is the primary key for this table. The title is the name of the movie. Table: Users

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id | int |
| name | varchar |

user\_id is the primary key for this table.

Table: MovieRating

|  |  |
| --- | --- |
| Column Name | Type |
| movie\_id | int |
| user\_id | int |
| rating | int |
| created\_at | date |

(movie\_id, user\_id) is the primary key for this table.

This table contains the rating of a movie by a user in their review. created\_at is the user's review date.

Write an SQL query to:

* Find the name of the user who has rated the greatest number of movies. In case of a tie, return the lexicographically smaller user name.
* Find the movie name with the highest average rating in February 2020. In case of a tie, return the lexicographically smaller movie name.

The query result format is in the following example.

Input:

Movies table:

|  |  |
| --- | --- |
| movie\_id | title |
| 1 | Avengers |
| 2 | Frozen 2 |
| 3 | Joker |

Users table:

|  |  |
| --- | --- |
| user\_id | name |
| 1 | Daniel |
| 2 | Monica |
| 3 | Maria |
| 4 | James |

MovieRating table:

|  |  |  |  |
| --- | --- | --- | --- |
| movie\_id | user\_id | rating | created\_at |
| 1 | 1 | 3 | 2020-01-12 |
| 1 | 2 | 4 | 2020-02-11 |
| 1 | 3 | 2 | 2020-02-12 |
| 1 | 4 | 1 | 2020-01-01 |
| 2 | 1 | 5 | 2020-02-17 |
| 2 | 2 | 2 | 2020-02-01 |
| 2 | 3 | 2 | 2020-03-01 |
| 3 | 1 | 3 | 2020-02-22 |
| 3 | 2 | 4 | 2020-02-25 |

Output:

|  |
| --- |
| results |
| Daniel |
| Frozen 2 |

Explanation:

Daniel and Monica have rated 3 movies ("Avengers", "Frozen 2" and "Joker") but Daniel is smaller lexicographically.

Frozen 2 and Joker have a rating average of 3.5 in February but Frozen 2 is smaller lexicographically.

**Query**: SELECT

(SELECT name

FROM Users

WHERE user\_id = (

SELECT user\_id

FROM MovieRating

GROUP BY user\_id

ORDER BY COUNT(\*) DESC, name ASC

LIMIT 1

)) AS user\_name,

(SELECT title

FROM Movies

WHERE movie\_id = (

SELECT movie\_id

FROM MovieRating

WHERE created\_at >= '2020-02-01' AND created\_at < '2020-03-01'

GROUP BY movie\_id

ORDER BY AVG(rating) DESC, title ASC

LIMIT 1

)) AS movie\_name

FROM MovieRating

LIMIT 1;

**Q36.**

Table: Users

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |

id is the primary key for this table. name is the name of the user. Table: Rides

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| user\_id | int |
| distance | int |

id is the primary key for this table. user\_id is the id of the user who travelled the distance "distance".

Write an SQL query to report the distance travelled by each user.

Return the result table ordered by travelled\_distance in descending order, if two or more users travelled the same distance, order them by their name in ascending order.

The query result format is in the following example.

Input:

Users table:

|  |  |
| --- | --- |
| id | name |
| 1 | Alice |
| 2 | Bob |
| 3 | Alex |
| 4 | Donald |
| 7 | Lee |
| 13 | Jonathan |
| 19 | Elvis |

Rides table:

|  |  |  |
| --- | --- | --- |
| id | user\_id | distance |
| 1 | 1 | 120 |
| 2 | 2 | 317 |
| 3 | 3 | 222 |
| 4 | 7 | 100 |
| 5 | 13 | 312 |
| 6 | 19 | 50 |
| 7 | 7 | 120 |
| 8 | 19 | 400 |
| 9 | 7 | 230 |

Output:

|  |  |
| --- | --- |
| name | travelled\_distan ce |
| Elvis | 450 |
| Lee | 450 |
| Bob | 317 |
| Jonathan | 312 |
| Alex | 222 |
| Alice | 120 |
| Donald | 0 |

Explanation:

Elvis and Lee travelled 450 miles, Elvis is the top traveller as his name is alphabetically smaller than Lee.

Bob, Jonathan, Alex, and Alice have only one ride and we just order them by the total distances of the ride.

Donald did not have any rides, the distance travelled by him is 0.

**Query**: SELECT Users.name, COALESCE(SUM(Rides.distance), 0) AS travelled\_distance

FROM Users

LEFT JOIN Rides ON Users.id = Rides.user\_id

GROUP BY Users.id, Users.name

ORDER BY travelled\_distance DESC, Users.name ASC;

**Q37.**

Table: Employees

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |

id is the primary key for this table.

Each row of this table contains the id and the name of an employee in a company.

Table: EmployeeUNI

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| unique\_id | int |

(id, unique\_id) is the primary key for this table.

Each row of this table contains the id and the corresponding unique id of an employee in the company.

Write an SQL query to show the unique ID of each user, If a user does not have a unique ID replace just show null.

Return the result table in any order.

The query result format is in the following example.

Input:

Employees table:

|  |  |
| --- | --- |
| id | name |
| 1 | Alice |
| 7 | Bob |
| 11 | Meir |
| 90 | Winston |
| 3 | Jonathan |

EmployeeUNI table:

|  |  |
| --- | --- |
| id | unique\_id |
| 3 | 1 |
| 11 | 2 |
| 90 | 3 |

Output:

|  |  |
| --- | --- |
| unique\_id | name |
| null | Alice |
| null | Bob |
| 2 | Meir |
| 3 | Winston |
| 1 | Jonathan |

Explanation:

Alice and Bob do not have a unique ID, We will show null instead.

The unique ID of Meir is 2.

The unique ID of Winston is 3.

The unique ID of Jonathan is 1.

**Query**: SELECT e.name, eu.unique\_id

FROM Employees e

LEFT JOIN EmployeeUNI eu ON e.id = eu.id

ORDER BY e.name;

**Q38.**

Table: Departments

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |

id is the primary key of this table.

The table has information about the id of each department of a university. Table: Students

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |
| department\_id | int |

id is the primary key of this table.

The table has information about the id of each student at a university and the id of the department he/she studies at.

Write an SQL query to find the id and the name of all students who are enrolled in departments that no longer exist.

Return the result table in any order.

The query result format is in the following example.

Input:

Departments table:

|  |  |
| --- | --- |
| id | name |
| 1 | Electrical Engineering |
| 7 | Computer Engineering |
| 13 | Business Administration |

Students table:

|  |  |  |
| --- | --- | --- |
| id | name | department\_id |
| 23 | Alice | 1 |
| 1 | Bob | 7 |
| 5 | Jennifer | 13 |
| 2 | John | 14 |
| 4 | Jasmine | 77 |
| 3 | Steve | 74 |
| 6 | Luis | 1 |
| 8 | Jonathan | 7 |
| 7 | Daiana | 33 |
| 11 | Madelynn | 1 |

Output:

|  |  |
| --- | --- |
| id | name |
| 2 | John |
| 7 | Daiana |
| 4 | Jasmine |
| 3 | Steve |

Explanation:

John, Daiana, Steve, and Jasmine are enrolled in departments 14, 33, 74, and 77 respectively. Department 14, 33, 74, and 77 do not exist in the Departments table.

**Query**: SELECT s.id, s.name

FROM Students s

LEFT JOIN Departments d ON s.department\_id = d.id

WHERE d.id IS NULL;

**Q39.**

Table: Calls

|  |  |
| --- | --- |
| Column Name | Type |
| from\_id | int |
| to\_id | int |
| duration | int |

This table does not have a primary key, it may contain duplicates. This table contains the duration of a phone call between from\_id and to\_id. from\_id != to\_id

Write an SQL query to report the number of calls and the total call duration between each pair of distinct persons (person1, person2) where person1 < person2.

Return the result table in any order.

The query result format is in the following example.

Input:

Calls table:

|  |  |  |
| --- | --- | --- |
| from\_id | to\_id | duration |
| 1 | 2 | 59 |
| 2 | 1 | 11 |
| 1 | 3 | 20 |
| 3 | 4 | 100 |
| 3 | 4 | 200 |
| 3 | 4 | 200 |
| 4 | 3 | 499 |

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| person1 | person2 | call\_count | total\_duration |
| 1 | 2 | 2 | 70 |
| 1 | 3 | 1 | 20 |
| 3 | 4 | 4 | 999 |

Explanation:

Users 1 and 2 had 2 calls and the total duration is 70 (59 + 11).

Users 1 and 3 had 1 call and the total duration is 20.

Users 3 and 4 had 4 calls and the total duration is 999 (100 + 200 + 200 + 499).

**Query**: SELECT

CASE WHEN from\_id < to\_id THEN from\_id ELSE to\_id END AS person1,

CASE WHEN from\_id < to\_id THEN to\_id ELSE from\_id END AS person2,

COUNT(\*) AS call\_count,

SUM(duration) AS total\_duration

FROM Calls

GROUP BY person1, person2;

**Q40.**

Table: Prices

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| start\_date | date |
| end\_date | date |
| price | int |

(product\_id, start\_date, end\_date) is the primary key for this table.

Each row of this table indicates the price of the product\_id in the period from start\_date to end\_date. For each product\_id there will be no two overlapping periods. That means there will be no two intersecting periods for the same product\_id.

Table: UnitsSold

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| purchase\_date | date |
| units | int |

There is no primary key for this table, it may contain duplicates.

Each row of this table indicates the date, units, and product\_id of each product sold.

Write an SQL query to find the average selling price for each product. average\_price should be rounded to 2 decimal places.

Return the result table in any order.

The query result format is in the following example.

Input:

Prices table:

|  |  |  |  |
| --- | --- | --- | --- |
| product\_id | start\_date | end\_date | price |
| 1 | 2019-02-17 | 2019-02-28 | 5 |
| 1 | 2019-03-01 | 2019-03-22 | 20 |
| 2 | 2019-02-01 | 2019-02-20 | 15 |
| 2 | 2019-02-21 | 2019-03-31 | 30 |

UnitsSold table:

|  |  |  |
| --- | --- | --- |
| product\_id | purchase\_date | units |
| 1 | 2019-02-25 | 100 |
| 1 | 2019-03-01 | 15 |
| 2 | 2019-02-10 | 200 |
| 2 | 2019-03-22 | 30 |

Output:

|  |  |
| --- | --- |
| product\_id | average\_price |
| 1 | 6.96 |
| 2 | 16.96 |

Explanation:

Average selling price = Total Price of Product / Number of products sold.

Average selling price for product 1 = ((100 \* 5) + (15 \* 20)) / 115 = 6.96 Average selling price for product 2 = ((200 \* 15) + (30 \* 30)) / 230 = 16.96 **Q41.**

Table: Warehouse

|  |  |
| --- | --- |
| Column Name | Type |
| name | varchar |
| product\_id | int |
| units | int |

(name, product\_id) is the primary key for this table.

Each row of this table contains the information of the products in each warehouse. Table: Products

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| product\_name | varchar |
| Width | int |
| Length | int |
| Height | int |

product\_id is the primary key for this table.

Each row of this table contains information about the product dimensions (Width, Length, and Height) in feets of each product.

Write an SQL query to report the number of cubic feet of volume the inventory occupies in each warehouse.

Return the result table in any order.

The query result format is in the following example.

Input:

Warehouse table:

|  |  |  |
| --- | --- | --- |
| name | product\_id | units |
| LCHouse1 | 1 | 1 |
| LCHouse1 | 2 | 10 |
| LCHouse1 | 3 | 5 |
| LCHouse2 | 1 | 2 |
| LCHouse2 | 2 | 2 |
| LCHouse3 | 4 | 1 |

Products table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| product\_id | product\_name | Width | Length | Height |
| 1 | LC-TV | 5 | 50 | 40 |
| 2 | LC-KeyChain | 5 | 5 | 5 |
| 3 | LC-Phone | 2 | 10 | 10 |
| 4 | LC-T-Shirt | 4 | 10 | 20 |

Output:

|  |  |
| --- | --- |
| warehouse\_name | volume |
| LCHouse1 | 12250 |
| LCHouse2 | 20250 |
| LCHouse3 | 800 |

**Query**: SELECT

UnitsSold.product\_id,

ROUND(SUM(Prices.price \* UnitsSold.units) / SUM(UnitsSold.units), 2) AS average\_price

FROM

UnitsSold

JOIN

Prices ON UnitsSold.product\_id = Prices.product\_id

AND UnitsSold.purchase\_date >= Prices.start\_date

AND UnitsSold.purchase\_date <= Prices.end\_date

GROUP BY

UnitsSold.product\_id;

**Q42.**

Table: Sales

|  |  |
| --- | --- |
| Column Name | Type |
| sale\_date | date |
| fruit | enum |
| sold\_num | int |

(sale\_date, fruit) is the primary key for this table.

This table contains the sales of "apples" and "oranges" sold each day.

Write an SQL query to report the difference between the number of apples and oranges sold each day.

Return the result table ordered by sale\_date.

The query result format is in the following example.

Input:

Sales table:

|  |  |  |
| --- | --- | --- |
| sale\_date | fruit | sold\_num |
| 2020-05-01 | apples | 10 |
| 2020-05-01 | oranges | 8 |
| 2020-05-02 | apples | 15 |
| 2020-05-02 | oranges | 15 |
| 2020-05-03 | apples | 20 |
| 2020-05-03 | oranges | 0 |
| 2020-05-04 | apples | 15 |
| 2020-05-04 | oranges | 16 |

Output:

|  |  |
| --- | --- |
| sale\_date | diff |
| 2020-05-01 | 2 |
| 2020-05-02 | 0 |
| 2020-05-03 | 20 |
| 2020-05-04 | -1 |

Explanation:

Day 2020-05-01, 10 apples and 8 oranges were sold (Difference 10 - 8 = 2).

Day 2020-05-02, 15 apples and 15 oranges were sold (Difference 15 - 15 = 0).

Day 2020-05-03, 20 apples and 0 oranges were sold (Difference 20 - 0 = 20).

Day 2020-05-04, 15 apples and 16 oranges were sold (Difference 15 - 16 = -1).

**Query**: SELECT sale\_date, SUM(CASE WHEN fruit = 'apples' THEN sold\_num ELSE -sold\_num END) AS diff

FROM Sales

GROUP BY sale\_date

ORDER BY sale\_date;

**Q43.**

Table: Activity

|  |  |
| --- | --- |
| Column Name | Type |
| player\_id | int |
| device\_id | int |
| event\_date | date |
| games\_played | int |

(player\_id, event\_date) is the primary key of this table.

This table shows the activity of players of some games.

Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on someday using some device.

Write an SQL query to report the fraction of players that logged in again on the day after the day they first logged in, rounded to 2 decimal places. In other words, you need to count the number of players that logged in for at least two consecutive days starting from their first login date, then divide that number by the total number of players.

The query result format is in the following example.

Input:

Activity table:

|  |  |  |  |
| --- | --- | --- | --- |
| player\_id | device\_id | event\_date | games\_played |
| 1 | 2 | 2016-03-01 | 5 |
| 1 | 2 | 2016-03-02 | 6 |
| 2 | 3 | 2017-06-25 | 1 |
| 3 | 1 | 2016-03-02 | 0 |
| 3 | 4 | 2018-07-03 | 5 |

Output:

|  |
| --- |
| fraction |
| 0.33 |

Explanation:

Only the player with id 1 logged back in after the first day he had logged in so the answer is 1/3 = 0.33

**Query:** SELECT ROUND(COUNT(DISTINCT player\_id) / COUNT(DISTINCT CASE WHEN DATE\_ADD(event\_date, INTERVAL 1 DAY) IN (SELECT DISTINCT event\_date FROM Activity) THEN player\_id END), 2) AS fraction

FROM Activity;

**Q44.**

Table: Employee

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |
| department | varchar |
| managerId | int |

id is the primary key column for this table.

Each row of this table indicates the name of an employee, their department, and the id of their manager.

If managerId is null, then the employee does not have a manager. No employee will be the manager of themself.

Write an SQL query to report the managers with at least five direct reports.

Return the result table in any order.

The query result format is in the following example.

Input:

Employee table:

|  |  |  |  |
| --- | --- | --- | --- |
| id | name | department | managerId |
| 101 | John | A | None |
| 102 | Dan | A | 101 |
| 103 | James | A | 101 |
| 104 | Amy | A | 101 |
| 105 | Anne | A | 101 |
| 106 | Ron | B | 101 |

Output:

|  |
| --- |
| name |
| John |

**Query**: SELECT e1.name AS name

FROM Employee e1

WHERE e1.id IN (

SELECT e2.managerId

FROM Employee e2

GROUP BY e2.managerId

HAVING COUNT(\*) >= 5

);

**Q45.**

Table: Student

|  |  |
| --- | --- |
| Column Name | Type |
| student\_id | int |
| student\_name | varchar |
| gender | varchar |
| dept\_id | int |

student\_id is the primary key column for this table.

dept\_id is a foreign key to dept\_id in the Department tables.

Each row of this table indicates the name of a student, their gender, and the id of their department. Table: Department

|  |  |
| --- | --- |
| Column Name | Type |
| dept\_id | int |
| dept\_name | varchar |

dept\_id is the primary key column for this table.

Each row of this table contains the id and the name of a department.

Write an SQL query to report the respective department name and number of students majoring in each department for all departments in the Department table (even ones with no current students). Return the result table ordered by student\_number in descending order. In case of a tie, order them by dept\_name alphabetically.

The query result format is in the following example.

Input:

Student table:

|  |  |  |  |
| --- | --- | --- | --- |
| student\_id | student\_name | gender | dept\_id |
| 1 | Jack | M | 1 |
| 2 | Jane | F | 1 |
| 3 | Mark | M | 2 |

Department table:

|  |  |
| --- | --- |
| dept\_id | dept\_name |
| 1 | Engineering |
| 2 | Science |
| 3 | Law |

Output:

|  |  |
| --- | --- |
| dept\_name | student\_numbe  r |
| Engineering | 2 |
| Science | 1 |
| Law | 0 |

**Query**: SELECT d.dept\_name AS dept\_name, COUNT(s.student\_id) AS student\_number

FROM Department d

LEFT JOIN Student s ON d.dept\_id = s.dept\_id

GROUP BY d.dept\_name

ORDER BY student\_number DESC, dept\_name ASC;

**Q46.**

Table: Customer

|  |  |
| --- | --- |
| Column Name | Type |
| customer\_id | int |
| product\_key | int |

There is no primary key for this table. It may contain duplicates. product\_key is a foreign key to the Product table. Table: Product

|  |  |
| --- | --- |
| Column Name | Type |
| product\_key | int |

product\_key is the primary key column for this table.

Write an SQL query to report the customer ids from the Customer table that bought all the products in the Product table.

Return the result table in any order.

The query result format is in the following example.

Input:

Customer table:

|  |  |
| --- | --- |
| customer\_id | product\_key |
| 1 | 5 |
| 2 | 6 |
| 3 | 5 |
| 3 | 6 |
| 1 | 6 |

Product table:

|  |
| --- |
| product\_key |
| 5 |
| 6 |

Output:

|  |
| --- |
| customer\_id |
| 1 |
| 3 |

Explanation:

The customers who bought all the products (5 and 6) are customers with IDs 1 and 3.\ **Q47.**

Table: Project

|  |  |
| --- | --- |
| Column Name | Type |
| project\_id | int |
| employee\_id | int |

(project\_id, employee\_id) is the primary key of this table. employee\_id is a foreign key to the Employee table.

Each row of this table indicates that the employee with employee\_id is working on the project with project\_id.

Table: Employee

|  |  |
| --- | --- |
| Column Name | Type |
| employee\_id | int |
| name | varchar |
| experience\_yea rs | int |

employee\_id is the primary key of this table.

Each row of this table contains information about one employee.

Write an SQL query that reports the most experienced employees in each project. In case of a tie, report all employees with the maximum number of experience years.

Return the result table in any order.

The query result format is in the following example.

Input:

Project table:

|  |  |
| --- | --- |
| project\_id | employee\_id |
| 1 | 1 |
| 1 | 2 |
| 1 | 3 |
| 2 | 1 |
| 2 | 4 |

Employee table:

|  |  |  |
| --- | --- | --- |
| employee\_id | name | experience\_yea rs |
| 1 | Khaled | 3 |
| 2 | Ali | 2 |
| 3 | John | 3 |
| 4 | Doe | 2 |

Output:

|  |  |
| --- | --- |
| project\_id | employee\_id |
| 1 | 1 |
| 1 | 3 |
| 2 | 1 |

Explanation:

Both employees with id 1 and 3 have the most experience among the employees of the first project. For the second project, the employee with id 1 has the most experience.

**Query**: SELECT customer\_id

FROM Customer

GROUP BY customer\_id

HAVING COUNT(DISTINCT product\_key) = (SELECT COUNT(DISTINCT product\_key) FROM Product);

**Q48.**

Table: Books

|  |  |
| --- | --- |
| Column Name | Type |
| book\_id | int |
| name | varchar |
| available\_from | date |

book\_id is the primary key of this table. Table: Orders

|  |  |
| --- | --- |
| Column Name | Type |
| order\_id | int |
| book\_id | int |
| quantity | int |
| dispatch\_date | date |

order\_id is the primary key of this table. book\_id is a foreign key to the Books table.

Write an SQL query that reports the books that have sold less than 10 copies in the last year, excluding books that have been available for less than one month from today. Assume today is 2019-06-23.

Return the result table in any order.

The query result format is in the following example.

Input:

Books table:

|  |  |  |
| --- | --- | --- |
| book\_id | name | available\_from |
| 1 | "Kalila And  Demna" | 2010-01-01 |
| 2 | "28 Letters" | 2012-05-12 |
| 3 | "The Hobbit" | 2019-06-10 |
| 4 | "13 Reasons  Why" | 2019-06-01 |
| 5 | "The Hunger  Games" | 2008-09-21 |

**Query**: SELECT b.book\_id, b.name

FROM Books b

LEFT JOIN Orders o ON b.book\_id = o.book\_id

WHERE o.dispatch\_date >= DATE\_SUB('2019-06-23', INTERVAL 1 YEAR)

AND b.available\_from <= DATE\_SUB('2019-06-23', INTERVAL 1 MONTH)

GROUP BY b.book\_id, b.name

HAVING SUM(o.quantity) < 10 OR SUM(o.quantity) IS NULL;

**Q49.**

Table: Enrollments

|  |  |
| --- | --- |
| Column Name | Type |
| student\_id | int |
| course\_id | int |
| grade | int |

(student\_id, course\_id) is the primary key of this table.

Write a SQL query to find the highest grade with its corresponding course for each student. In case of a tie, you should find the course with the smallest course\_id.

Return the result table ordered by student\_id in ascending order. The query result format is in the following example.

Input:

Enrollments table:

|  |  |  |
| --- | --- | --- |
| student\_id | course\_id | grade |
| 2 | 2 | 95 |
| 2 | 3 | 95 |
| 1 | 1 | 90 |
| 1 | 2 | 99 |
| 3 | 1 | 80 |
| 3 | 2 | 75 |
| 3 | 3 | 82 |

Output:

|  |  |  |
| --- | --- | --- |
| student\_id | course\_id | grade |
| 1 | 2 | 99 |
| 2 | 2 | 95 |
| 3 | 3 | 82 |

**Query**: SELECT student\_id, course\_id, grade

FROM (

SELECT student\_id, course\_id, grade,

ROW\_NUMBER() OVER (PARTITION BY student\_id ORDER BY grade DESC, course\_id ASC) AS rank

FROM Enrollments

) AS ranked

WHERE rank = 1

ORDER BY student\_id;

**Q50.**

Table: Teams

|  |  |
| --- | --- |
| Column Name | Type |
| team\_id | int |
| team\_name | varchar |

team\_id is the primary key of this table.

Each row of this table represents a single football team. Table: Matches

|  |  |
| --- | --- |
| Column Name | Type |
| match\_id | int |
| host\_team | int |
| guest\_team | int |
| host\_goals | int |
| guest\_goals | int |

match\_id is the primary key of this table.

Each row is a record of a finished match between two different teams.

Teams host\_team and guest\_team are represented by their IDs in the Teams table (team\_id), and they scored host\_goals and guest\_goals goals, respectively.

The winner in each group is the player who scored the maximum total points within the group. In the case of a tie, the lowest player\_id wins.

Write an SQL query to find the winner in each group.

Return the result table in any order.

The query result format is in the following example.

Input:

Players table:

|  |  |
| --- | --- |
| player\_id | group\_id |
| 15 | 1 |
| 25 | 1 |
| 30 | 1 |
| 45 | 1 |
| 10 | 2 |
| 35 | 2 |
| 50 | 2 |
| 20 | 3 |
| 40 | 3 |

Matches table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| match\_id | first\_player | second\_player | first\_score | second\_score |
| 1 | 15 | 45 | 3 | 0 |
| 2 | 30 | 25 | 1 | 2 |
| 3 | 30 | 15 | 2 | 0 |
| 4 | 40 | 20 | 5 | 2 |
| 5 | 35 | 50 | 1 | 1 |

Output:

|  |  |
| --- | --- |
| group\_id | player\_id |
| 1 | 15 |
| 2 | 35 |
| 3 | 40 |

**Query**: WITH Points AS (

SELECT player\_id, group\_id, SUM(points) AS total\_points

FROM (

SELECT match\_id, first\_player AS player\_id, group\_id,

CASE

WHEN first\_score > second\_score THEN 3

WHEN first\_score = second\_score THEN 1

ELSE 0

END AS points

FROM Matches

UNION ALL

SELECT match\_id, second\_player AS player\_id, group\_id,

CASE

WHEN second\_score > first\_score THEN 3

WHEN second\_score = first\_score THEN 1

ELSE 0

END AS points

FROM Matches

) AS match\_points

GROUP BY player\_id, group\_id

),

RankedPoints AS (

SELECT player\_id, group\_id, total\_points,

ROW\_NUMBER() OVER (PARTITION BY group\_id ORDER BY total\_points DESC, player\_id ASC) AS rank

FROM Points

)

SELECT group\_id, player\_id

FROM RankedPoints

WHERE rank = 1;